CS2030s Code Library

github.com/reidenong/cheatsheets, AY23/24 S1

Order for class modifiers:

1. `public protected private`
2. `abstract static final`

PE1

`Array<T>` Generic Array

```
class Array <T> {
  private T[] array;

  Array(int size) {
    // The only way we can put an object into the array is through
    // the method set() and we only put an object of type T inside.
    // So it is safe to cast `Object[]` to `T[]`.
    @SuppressWarnings("unchecked")
    T[] a = (T[]) new Object[size];
    this.array = a;
  }

  public void set(int index, T item) {
    this.array[index] = item;
  }

  public T get(int index) {
    return this.array[index];
  }

  public void copyFrom(Array << ? extends T > src) {
    int len = Math.min(this.array.length, src.array.length);
    for (int i = 0; i < len; i++) {
      this.set(i, src.get(i));
    }
  }

  public void copyTo(Array << ? super T > dest) {
    int len = Math.min(this.array.length, dest.array.length);
    for (int i = 0; i < len; i++) {
      dest.set(i, this.get(i));
    }
  }
}
```

Implementing `Comparable<T>`

```
class Packet implements Comparable < Packet > {
  private String message;

  public Packet(String message) {
    this.message = message;
  }

  @Override
  public String toString() {
    return this.message;
  }

  public int compareTo(Packet packet2) {
    if (this.message.length() == packet2.message.length()) {
      return 0;
    } else if (this.message.length() < packet2.message.length()) {
      return 1;
    } else {
      return -1;
    }
  }
}
```

Implementing a `Comparable` Generic class

```
public class Buffer < T extends Comparable <T>> {
  private T[] messages;
  private int endIndex;

  public Buffer(int size) {
    // The only way we can put an object into array is through the method
    // send and we only put Object of type T inside.
    // Thus it is safe to cast 'Object[]' to 'T[]'.
    @SuppressWarnings("unchecked")
    T[] temp = (T[]) new Comparable<?>[size];
    this.messages = temp;
    this.endIndex = 0;
  }
  ... ...
}
```

PE2

Implementing a `Try<T>` Monad

```
package cs2030s.fp;

public abstract class Try<T> {
  public static <T> Try<T> of(Producer<? extends T> prod) {
    try {
      return new Success<>(prod.produce());
    } catch (Throwable exc) {
      return new Failure<>(exc);
    }
  }

  public static <T> Try<T> success(T value) {
    return new Success<T>(value);
  }

  public static <T> Try<T> failure(Throwable exc) {
    return new Failure<>(exc);
  }

  // abstract methods
  public abstract T get() throws Throwable;
  public abstract <R> Try<R> map(Transformer<? super T, ? extends R> fn);
  public abstract <R> Try<R> flatMap(Transformer<? super T,
                                     ? extends Try<? extends R>> fn);
  public abstract Try<T> onFailure(Consumer<? super Throwable> cons);
  public abstract Try<T> recover(Transformer<? super Throwable,
                                 ? extends T> fn);

  private static class Success<T> extends Try<T> {
    private T value;

    public Success(T value) {
      this.value = value;
    }

    @Override
    public T get() throws Throwable {
      return this.value;
    }

    @Override
    public <R> Try<R> map(Transformer<? super T, ? extends R> fn) {
      return Try.of(() -> fn.transform(this.value));
    }

    @Override
    public <R> Try<R> flatMap(Transformer<? super T,
                              ? extends Try<? extends R>> fn) {
      return Try.of(() -> fn.transform(this.value).get());
    }

    @Override
    public Try<T> onFailure(Consumer<? super Throwable> cons) {
      return this;
    }

    @Override
    public Try<T> recover(Transformer<? super Throwable, ? extends T> fn)
{
      return this;
    }

    @Override
    public boolean equals(Object obj) {
      if (this == obj) {
        return true;
      }
      if (obj instanceof Success<?>) {
        Success<?> success = (Success<?>) obj;
        if (this.value == null) {
          return success.value == null;
        } else {
          return this.value.equals(success.value);
        }
      }
      return false;
    }
  }
  private static class Failure<T> extends Try<T> {
    private Throwable exc;

    public Failure(Throwable exc) {
      this.exc = exc;
    }

    @Override
    public T get() throws Throwable {
      throw this.exc;
    }
```

```
    @Override
    public <R> Try<R> map(Transformer<? super T, ? extends R> fn) {
      return this.self();
    }

    @Override
    public <R> Try<R> flatMap(Transformer<? super T,
                              ? extends Try<? extends R>> fn) {
      return this.self();
    }

    @Override
    public Try<T> onFailure(Consumer<? super Throwable> cons) {
      try {
        cons.consume(this.exc);
      } catch (Throwable exc) {
        return Try.failure(exc);
      }
      return this.self();
    }

    @Override
    public Try<T> recover(Transformer<? super Throwable, ? extends T>fn){
      return Try.of(() -> fn.transform(this.exc));
    }
    // Used to combine all the @SuppressWarnings into one location
    private <R> Try<R> self() {
      @SuppressWarnings("unchecked")
      Try<R> res = (Try<R>) this;
      return res;
    }

    @Override
    public boolean equals(Object obj) {
      if (this == obj) {
        return true;
      }
      if (obj instanceof Failure<?>) {
        Failure<?> failure = (Failure<?>) obj;
        return this.exc.toString().equals(failure.exc.toString());
      }
      return false;
    }
  }
}
```

Stream Snippets

Fibonacci

```
  Stream<Integer> fibonacci(int n) {}
    return Stream.iterate(new int[]{1, 1},
                          x -> new int[]{x[1], x[0] + x[1]})
                 .map(x -> x[0])
                 .limit(n)
  }
```

Breaking Stream into Groupings

```
  import java.util.concurrent.atomic.AtomicInteger

  public Stream < Integer > groupSum(int groupSize) {
    AtomicInteger counter = new AtomicInteger(0);
    return Stream.of(1, 2, 3, 4, 5, 6, 7, 8, 9)
        .collect(Collectors.groupingBy(
                  x -> counter.getAndIncrement() / groupSize
                  ))
        .entrySet()
        .stream()
        .map(Map.Entry::getValue)
        .map(x -> x.stream().reduce(0, (sub, curr) -> sub + curr))
        .forEach(System.out::println);
  }
  // Prints 6; 15; 24;
```

`Maybe<T>`

Creation

`static <T> Maybe<T> of(T val)`	Creates a `Maybe<T>` object with a value of type `T` if value is not `null`. Otherwise, returned the shared instance of `None<?>`.
`static <T> Maybe<T> some(T val)`	Creates a `Maybe<T>` object with a value of type `T` where value may be `null`.
`static <T> Maybe<T> none()`	Guaranteed to return shared `None<T>`.

Intermediate Operations

`<U> Maybe<U> map (Transformer<? super T, ? extends U> fn)`	`Maybe`: Creates a new instance of `Maybe` by applying the transformer `fn` to the content and wrapping it in `Maybe`. If result is `null`, return shared instance of `None<?>`. `None`: Returns `None<T>`
`<U> Maybe<U> flatMap (Transformer<? super T, ? extends Maybe<? extends U>> fn)`	`Maybe`: Create a new instance of `Maybe` by applying the transformer `fn` to the content without wrapping. . `None`: Returns `None<T>`
`Maybe<T> filter (Predicate<? super T> pred)`	`Maybe`: Returns the current instance of `Maybe` if the content satisfies the predicate `pred`. Otherwise, return shared instance of `None<?>`. `None`: Returns `None<T>`

Terminal Operations

`T orElse(Producer<? extends T> prod)`	`Maybe`: Returns the content (even if it is null) `None`: Returns the value produced by `prod`
`void ifPresent(Consumer<? super T> cons)`	`Maybe`: Pass the content to consumer `cons` `None`: Do Nothing
`String toString()`	Returns the String representation of `Maybe`.
`boolean equals(Object obj)`	`Maybe`: Returns `true` if the content is equal to the content of `obj`. `None`: Returns `true` if `obj` is also `None<?>`

`Lazy<T>`

Creation

`static <T> Lazy<T> of(T val)`	Creates a `Lazy<T>` object with the given content `val` already evaluated.
`static <T> Lazy<T> of(Producer<? extends T> prod)`	Creates a `Lazy<T>` with the content not yet evaluated.

Intermediate Operations

`<U> Lazy<U> map (Transformer<? super T, ? extends U> fn)`	Lazily maps the content using the given transformer.
`<U> Lazy<U> flatMap (Transformer<? super T, ? extends Lazy<? extends U>> fn)`	Lazily creates a new instance of `Lazy` by applying the transformer `fn` to the content without wrapping.
`Lazy<Boolean> filter (BooleanCondition<? super T> pred)`	Lazily test if the value passes the test or not and returns a `Lazy<Boolean>` to indicate the result.

Terminal Operations

`T get()`	If content is not evaluated, evaluate it and return the content. Otherwise, return the content.
`boolean equals()`	Forces evaluation of content. Returns `true` if the content is equal to the content of `obj`.